Fixed Block Compression Boosting in FM-Indexes[*]

Juha Kärkkäinen[1] and Simon J. Puglisi[2]

[1] Department of Computer Science, University of Helsinki, Finland
juha.karkkainen@cs.helsinki.fi

[2] Department of Informatics, King's College London, London, United Kingdom
simon.puglisi@kcl.ac.uk

Abstract. A compressed full-text self-index occupies space close to that of the compressed text and simultaneously allows fast pattern matching and random access to the underlying text. Among the best compressed self-indexes, in theory and in practice, are several members of the FM-index family. In this paper, we describe new FM-index variants that combine nice theoretical properties, simple implementation and improved practical performance. Our main result is a new technique called *fixed block compression boosting*, which is a simpler and faster alternative to optimal compression boosting and implicit compression boosting used in previous FM-indexes.

1 Introduction

A compressed full-text self-index [13] of a text string T is a data structure that stores T in a compressed form that allows fast random access to T and also supports fast pattern matching queries. We focus here on the *count* query that, given a pattern string P, returns the number of occurrences of P in T. [3] Many of the best compressed self-indexes, in theory and in practice, belong to the FM-index family originating from the FM-index of Ferragina and Manzini [5]. In particular, they combine good compression with fast count queries [6, 11, 4, 2]. In this paper, we describe new variants of the FM-family achieving even better compression and faster count queries.

The main components of most FM-indexes are:

- The *Burrows–Wheeler transform* (BWT) [1]: an invertible permutation of the text T. A procedure called *backward search* [5] turns a count query on T into a sequence of *rank* queries on the BWT.
- The *wavelet tree* [7]: a representation of the BWT that turns a BWT rank query into a sequence of rank queries on bitvectors.
- A bitvector *rank index*, which supports fast rank queries on bitvectors.

The total length of standard wavelet tree bitvectors is equal to the size of the original, uncompressed text in bits. All other data structures can be fitted in less space: asymptotically less in theory, and significantly less in practice. Basic zero-order compression is achieved either with compressed bitvector rank structures, such as RRR [15], or Huffman-shaped wavelet trees [8]. For higher order compression, we can use a technique called *compression boosting* [3, 6], where the BWT is partitioned into blocks of varying sizes based on the context of symbols in T, and there is a separate, zero-order compressed wavelet tree for each block. An optimal partitioning into context blocks can be found in linear time [3].

---

[3] Our indexes support other common queries such as *locate* and *extract*, but the algorithmic and implementation issues in engineering them are quite different and outside the scope of this paper.

Our main result is a technique called *fixed block compression boosting*. It is similar to context block boosting, but divides the BWT into blocks of fixed sizes without any regard to the symbol contexts. Such a division is inoptimal, but we show that it cannot be much worse than the optimal one. What we gain is simpler and faster data structures. The difference is particularly dramatic in the construction phase.

The RRR-structure for compressed bitvector rank [15] divides the bitvectors into small blocks of fixed sizes. Mäkinen and Navarro [11] show that this achieves a similar compression boosting effect without any explicit division of the BWT. This is called *implicit compression boosting*. Their analysis of the effect of fixed blocks inspired our analysis, but the extension from small blocks on bitvectors to larger blocks and larger alphabets is non-trivial.

There are implementations of FM-indexes without any compression boosting [4], with optimal context block boosting [4], and with implicit boosting [2]. Fixed block boosting has practical advantages over all these, which we demontrate experimentally.

2 Basic Algorithmic Machinery

Let $T = T[0..n-1] = T[0]T[1]\ldots T[n-1]$ be a string of n symbols or characters drawn from an alphabet $\Sigma = \{0, 1, .., \sigma - 1\}$. We assume that $T[n-1] = 0$ and 0 does not appear anywhere else in T. In the examples, we use '\$' to denote 0 and letters to denote other symbols.

For any $i \in 0..n-1$, the string $T[i..n-1]T[0..i-1]$ is a *rotation* of T. Let $\mathcal{M}$ be the $n \times n$ matrix whose rows are all the rotations of T in lexicographic order. Let F be the first and L the last column of $\mathcal{M}$, both taken to be strings of length n. The string L is the **Burrows–Wheeler transform** of T. An example is given in Fig. 1. Note that F and L are permutations of T.

$$
\begin{array}{c c}
F & L \\
\fbox{\$} \; \text{B A N A N} \; \fbox{A} \\
\fbox{A} \; \text{\$ B A N A} \; \fbox{N} \\
\fbox{A} \; \text{N A \$ B A} \; \fbox{N} \\
\fbox{A} \; \text{N A N A \$} \; \fbox{B} \\
\fbox{B} \; \text{A N A N A} \; \fbox{\$} \\
\fbox{N} \; \text{A \$ B A N} \; \fbox{A} \\
\fbox{N} \; \text{A N A \$ B} \; \fbox{A}
\end{array}
$$

Fig. 1: BWT matrix $\mathcal{M}$ for text $T = \texttt{BANANA\$}$.

The FM-family of compressed text self-indexes is based on a procedure called **backward search**, which finds the range of rows in $\mathcal{M}$ that begin with a given pattern P. This range represents the occurrences of P in T. Fig. 2 shows how backward search is used for implementing the count query. In the algorithm, $C[c]$ is the position of the first occurrence of the symbol c in F, and the function rank_L is defined as

$$
\mathrm{rank}_L(c, j) \equiv \big|\{i \mid i < j \text{ and } L[i] = c\}\big|
$$

The main difference between the members of the FM-family is how they implement the rank_L-function. The best ones use wavelet trees.

A wavelet tree of a string X over an alphabet Σ is a binary tree with leaves labelled by the symbols of Σ. Each node v is associated with the subsequence of X consisting of those symbols

Algorithm FM-Count($P[0..m-1]$)
1: $b \leftarrow 0; e \leftarrow n$
2: **for** $i \leftarrow m-1$ **downto** 0 **do**
3:     $c \leftarrow P[i]$
4:     $b \leftarrow C[c] + \mathrm{rank}_L(c,b)$
5:     $e \leftarrow C[c] + \mathrm{rank}_L(c,e)$
6:     **if** $b = e$ **then break**   //The range is empty
7: **return** $e - b$ //The range is $b..e-1$

Fig. 2: Counting pattern occurrences using backward search.

that appear in the subtree rooted at v. The associated strings are not stored; instead each internal node v stores a bitvector $B(v)$ that tells for each character in the associated string whether it is in the left or right subtree of v. Fig. 3 shows examples of the two commonly used variants of wavelet trees, the balanced and the Huffman-shaped.

The balanced wavelet tree is easy to implement with low overhead. The total length of the bitvectors is $|X|\lceil\log|\Sigma|\rceil$, which is exactly the length of X in bits using the standard representation. On the other hand, the Huffman-shaped wavelet tree (HWT) is the one that minimizes the total length of the bitvectors, which equals the size of the Huffman compressed string X.
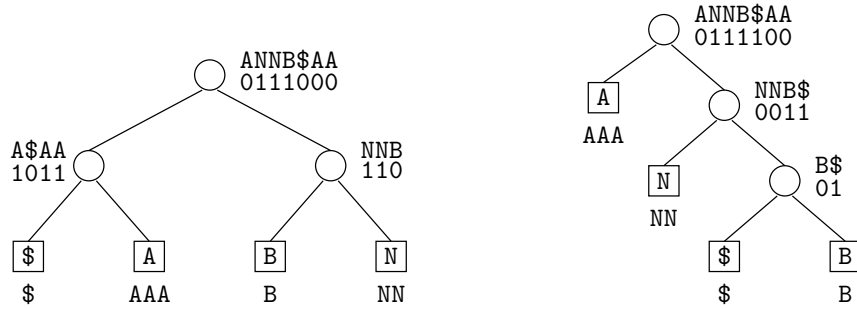
Fig. 3: Balanced (left) and Huffman-shaped (right) wavelet trees.

A rank query $\mathrm{rank}_X(c,r)$ over a wavelet tree is evaluated by a traversal from the root to the leaf labelled by c, as shown in Fig. 4. The procedure involves rank queries over the bitvectors stored on the root-to-leaf path.

There are many data structures for representing bitvectors so that rank queries can be answered efficiently [14, 16, 2]. They can be divided into two main categories. Uncompressing techniques leave the bitvector intact but use a small (usually sublinear) data structure on top of it. Compressing techniques compress the bitvector as well as prepare it for rank queries.

3

```
Algorithm WT-Rank(c, r)
1: v ← root; q ← r
2: while v is not a leaf do
3:     if c is in the left subtree of v then
4:         q ← q − rank_{B(v)}(1, q)
5:         v ← leftchild(v)
6:     else
7:         q ← rank_{B(v)}(1, q)
8:         v ← rightchild(v)
9: return q
```

Fig. 4: Rank operation using a wavelet tree.

3 Compression Boosting

Recall that T is a string of length n over an alphabet Σ of size σ. For each $c \in \Sigma$, let n_c denote the number of occurrences of c in T. The zero-order empirical entropy [12] of T is

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} = \log n - \frac{1}{n} \sum_{c \in \Sigma} n_c \log n_c. \tag{1}$$

Let n_w be the number of occurrences of a string w in T, and let $T|w$ be the subsequence of T consisting of those characters that appear in the *(right) context* w, i.e., that are immediately followed by w. Here T is taken to be a cyclic string, so that each character has a context of every length. The kth order empirical entropy is

$$H_k(T) = \sum_{w \in \Sigma^k} \frac{n_w}{n} H_0(T|w) \ .$$

The value $nH_k(T)$ represents a lower bound on the number of bits needed to encode T by any compressor that considers a context of size at most k when encoding a symbol. Note that $H_{k+1}(T) \leq H_k(T)$ for all k.

A remarkable property of L, the BWT of T, is that $T|w$ is a contiguous substring of L for any w; we call the substring the w-context block of L. For example, if $T = \texttt{BANANA\$}$, then $T|\texttt{A} = \texttt{NNB} = L[1..3]$ (see Fig. 1). Thus we get the following result.

Lemma 1 ([12]) *For any $k \geq 0$, there exists a partitioning of $L_1 L_2 \cdots L_\ell = L$ of the BWT L of T into $\ell \leq \sigma^k$ blocks so that*

$$\sum_{i=1}^{\ell} |L_i| H_0(L_i) = nH_k(T) \ .$$

In other words, by compressing each BWT block to zero-order entropy level, we obtain k^{th} order entropy compression for the whole text. This is called *compression boosting* [3].

The space requirement of an FM-index is usually dominated by the wavelet tree bitvectors. The total length of the bitvectors in the balanced wavelet tree of L is $n\lceil \log \sigma \rceil$. Using a Huffman-shaped wavelet tree reduces this down to at most $n(H_0(T) + 1)$. An alternative way to achieve

zero-order compression is to use compressed bitvector rank indexes. For example, using a rank index of Raman, Raman and Rao (RRR) [15], the total size of the rank indexes (without HWT or boosting) is $nH_0(T) + o(n)\log\sigma$.

Compression boosting improves the $H_0(T)$ factor to $H_k(T)$ [6]: Divide the BWT into context blocks using context of length k and implement a separate wavelet tree for each block. There is an additional space overhead of $\mathcal{O}(\sigma\log n)$ bits per block from having many blocks and wavelet trees instead of just one. The total overhead is $o(n)$ for $k \leq ((1-\epsilon)\log_\sigma n) - 1$ and any constant $\epsilon > 0$.

It may not be optimal to use the same context length in all parts of L. Ferragina et al. [3] show how to find an optimal partitioning with varying context length in linear time. The resulting compression is at least as good as with *any* fixed k.

Mäkinen and Navarro [11] show that the boosting effect is achieved with the RRR bitvector rank index without any explicit context partitioning. This is called *implicit compression boosting*. First, they observe that instead of partitioning the BWT, we could partion the bitvectors and obtain the same boosting effect. Second, the RRR technique partitions the bitvectors into blocks of size $b = (\log n)/2$ and compresses each independently. The RRR partitioning is not optimal, but Mäkinen and Navarro show that the overhead due to the inoptimality is at most $2\sigma\ell b \leq \sigma^{k+1}\log n = o(n)$ under the assumptions mentioned above.

Theorem 2 ([6, 11]) *The FM-index either with explicit boosting and optimal partitioning [6] or with implicit boosting [11] can be implemented in $nH_k(T) + o(n)\log\sigma$ bits of space for any $k \leq ((1-\epsilon)\log_\sigma n) - 1$ and any constant $\epsilon > 0$.*

4 Fixed Block Compression Boosting

In this section, we show that the compression boosting effect can also be achieved with a partitioning into blocks of fixed sizes without any regard to symbol context.

Let $H(x,y) = |B|H_0(B)$, where B is a bitvector containing x 0's and y 1's. Let $|X|_c$ denote the number of occurrences of a symbol c in a string X. The following lemma shows what can happen to the total zero-order entropy when two strings are concatenated.

Lemma 3 *For any two strings X and Y over an alphabet Σ,*

$$0 \leq |XY|H_0(XY) - |X|H_0(X) - |Y|H_0(Y)$$
$$= H(|X|,|Y|) - \sum_{c\in\Sigma} H(|X|_c,|Y|_c) \leq H(|X|,|Y|) \leq |XY| .$$

Proof. The last two inequalities are trivial and the first is a standard application of Gibb's inequality. We will prove the equality part. For brevity, we write $x = |X|$, $y = |Y|$, $x_c = |X|_c$ and $y_c = |Y|_c$. Using (1), we can write the left-hand side terms as follows

$$(x+y)H_0(XY) = (x+y)\log(x+y) - \sum_{c\in\Sigma}(x_c+y_c)\log(x_c+y_c)$$

$$xH_0(X) = x\log x - \sum_{c\in\Sigma} x_c\log x_c$$

$$yH_0(Y) = y\log y - \sum_{c\in\Sigma} y_c\log y_c$$

5

and the right-hand side terms as follows

$$H(x,y) = (x+y)\log(x+y) - x\log x - y\log y$$
$$H(x_c, y_c) = (x_c + y_c)\log(x_c + y_c) - x_c\log x_c - y_c\log y_c$$

From this it is easy to see that the terms on both sides match. $\qquad\square$

In other words, the concatenation cannot reduce the total entropy, and the entropy can increase by at most one bit per character. Furthermore, the maximum increase happens only if the two strings have the same length and no common symbols.

Using the above lemma we can bound the increase in entropy when we switch from a context block partitioning to a fixed block partitioning.

Lemma 4 *Let $X_1 X_2 \cdots X_\ell = X$ be a string partitioned arbitrarily into ℓ blocks. Let $X_1^b X_2^b \cdots X_m^b = X$ be a partition of X into blocks of size at most b. Then*

$$\sum_{i=1}^{m} |X_i^b| H_0(X_i^b) \leq \sum_{i=1}^{\ell} |X_i| H_0(X_i) + (\ell - 1)b \ .$$

Proof. Consider a process, where we start with the first partitioning, add the split points of the second partitioning, and then remove the split points of the first partitioning (that are not split points in the second). By Lemma 3, adding split points cannot increase the total entropy, and removing each split point can increase the entropy by at most b bits. $\qquad\square$

If we assume the same number of blocks in the two partitionings, the very worst case increase in the entropy is $n - b$ bits. However, such a worst case is very unlikely and in practice the increase is much smaller.

If we set the block size to $b = \sigma(\log n)^2$, we obtain the following result.

Theorem 5 *The FM-index with explicit boosting and blocks of fixed sizes can be implemented in $H_k(T) + o(n)\log\sigma$ bits of space for any $k \leq ((1-\epsilon)\log_\sigma n) - 1$ and any constant $\epsilon > 0$.*

Proof. Using context block boosting with fixed context length k and RRR to compress the bitvectors, the size of the FM-index is $nH_k(T) + o(n)\log\sigma$ bits. When we switch from context blocks to fixed blocks, we must add two types of overhead. First, by Lemma 4, the total entropy increases by at most $\sigma^k b = \sigma^{k+1}(\log n)^2 = n^{1-\epsilon}(\log n)^2 = o(n)$ bits. Second, the space needed for everything else but the bitvector rank indexes is $\mathcal{O}(\sigma\log n)$ bits per block. In total, this is $\mathcal{O}(n/\log n) = o(n)$ bits. Thus the total increase in the size of the FM index is $o(n)$ bits. $\qquad\square$

Thus, we have the same theoretical result as with context block boosting or implicit boosting.

The advantages of fixed block boosting compared to context block boosting are:

- To compute $\text{rank}_L(c, r)$, we have to find the block containing the position r. With fixed blocks this is simpler and faster than with varying size context blocks.
- Computing the optimal partitioning is complicated and expensive in practice. With fixed blocks, construction is much simpler and faster.

Explicit boosting (with either context blocks or fixed blocks) enables faster queries than implicit boosting for the following reasons:

Table 1: Data sets used for empirical tests. For each type of data (DNA, XML, ENGLISH, SOURCE) a 100Mb file was used.

Data set name	σ	H_0	mean LCP
XML	97	5.23	44
DNA	16	1.98	31
ENGLISH	239	4.53	2,221
SOURCE	230	5.54	168

- Compressed bitvector rank indexes are slower than uncompressed ones by a significant constant factor. Explicit boosting can achieve higher order compression with Huffman-shaped wavelet trees allowing the use of the faster uncompressed rank indexes.
- With implicit boosting, i.e., with a single wavelet tree for the whole BWT, the average count query time for a pattern P is $\Theta(|P|\log\sigma)$ with a balanced wavelet tree and $\Theta(|P|H_0(T))$ with a HWT. With explicit boosting and HWTs, the average query time is reduced down to $\mathcal{O}(|P|H_k(T))$.

5 Experimental Results

To assess practical performance we used the files listed in Table 1[4]. All tests were conducted on a 3.0 GHz Intel Xeon CPU with 4Gb main memory and 1024K L2 Cache. The machine had no other significant CPU tasks running. The operating system was Fedora Linux running kernel 2.6.9. The compiler was g++ (gcc version 4.1.1) executed with the -O3 option. The times given were recorded with the C `getrusage` function. The memory requirements are sums of the sizes of all data structures as reported by the `sizeof` function.

We measured the following FM-Index variants:

- SSA [10] simply stores a single HWT for the whole L, consuming $nH_0 + o(n\log\sigma)$ bits of space. This is the fastest index according to experiments in both [2] and [4].
- SSA+RRR is the implicit compression boosting approach of Mäkinen and Navarro [11]. As with SSA it builds a single HWT of L, however the bitvectors of the wavelet tree are now stored in a RRR compressed rank data structure. This method was first implemented by Claude and Navarro [2].
- AFFMI [6] uses optimal context block boosting with a separate HWT for each block. The implementation we use is from [4].
- Fixed Block and Fixed Block+RRR are implementations of the new fixed block boosting technique that use, respectively, plain and RRR preprocessed HWTs to represent blocks.

Figure 5 shows the trade-off between index size and pattern counting time. Following the methodology of [2, 4] we report query times averaged over a large number of random patterns, extracted from the underlying text.

With the compressible texts (XML, SOURCE and ENGLISH) the fixed block indexes dominate the others in both space and time. On DNA, which is not very compressible, fixed block indexes are still small and fast, but the ranks stored at block boundaries are no longer paid for by compression and
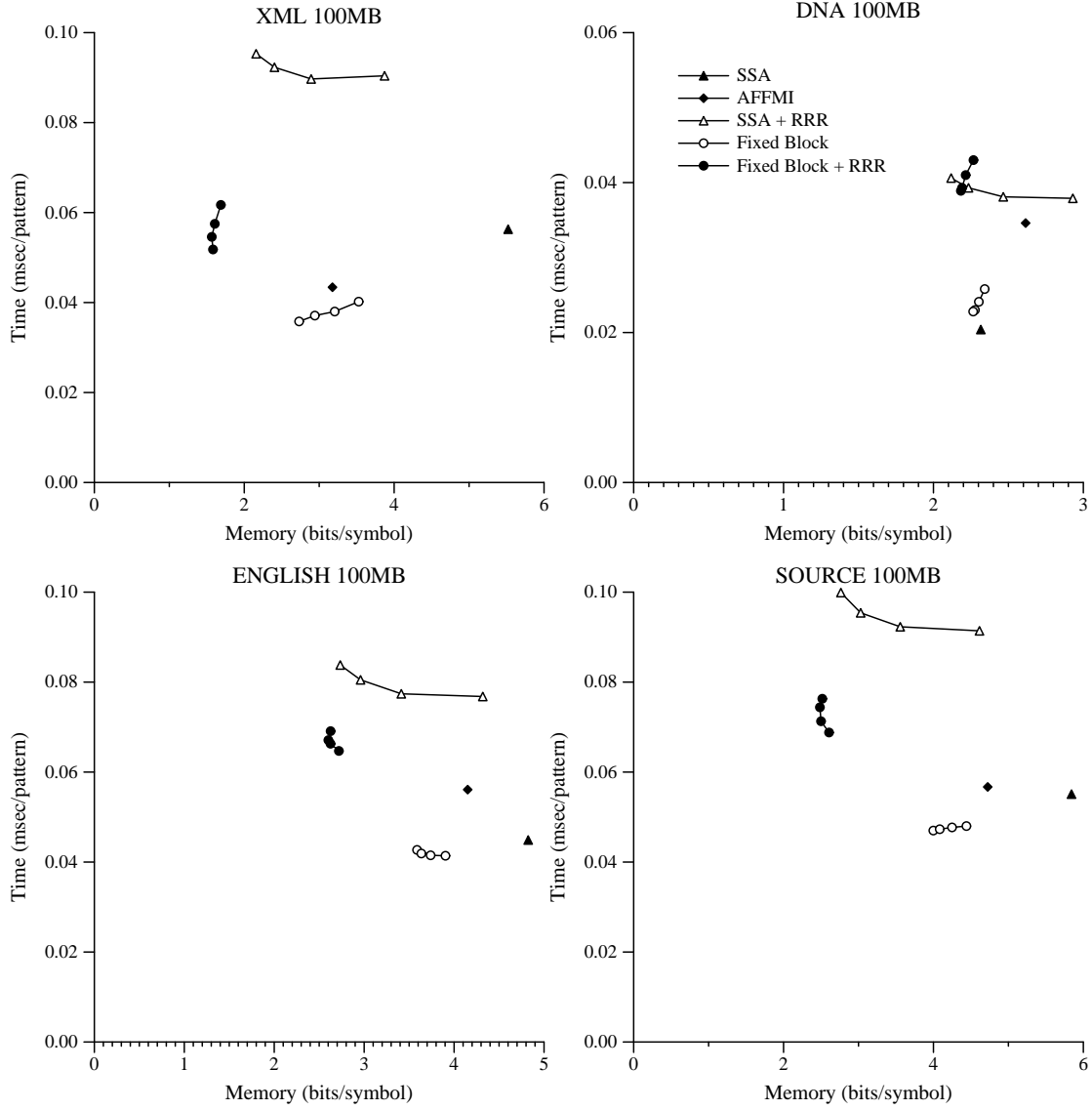

Fig. 5: Time-Space tradeoff for various self-indexes. Memory (abcissa) is the index size in bits per input symbol. Time (ordinate) is the average number of milliseconds taken to count pattern occurrences (averaged over 10^6 patterns).

the SSA+RRR, which does not need to store ranks at block boundaries, is the smallest index. The small alphabet of DNA means the single HWT of the SSA is shallow, making it fast.

The AFFMI, despite using optimal partitioning, is larger and slower than the fixed block indexes. AFFMI stores a bitvector marking the partitioning and issues a rank query on this bitvector to determine the appropriate wavelet tree to use at each step in the backward search process. This adds a significant time and space overhead which the fixed block approach avoids entirely.

6 Concluding Remarks

The indexes we have presented based on fixed block compression boosting are the most practical self-indexes to date, but we believe there is yet more room for improvement. Our current focus is on improving the RRR data structure to better exploit the structure of wavelet tree bitvectors produced by the BWT. We are also exploring an improved implementation of Huffman-shaped wavelet trees which use substantially less space, enabling smaller blocks and thus better compression.

A virtue of fixed block compression boosting our experiments have not touched on is construction, which is easier with fixed blocks. The final phase of indexing, where the BWT is turned into an FM-index, now requires only $nH_k + o(n)\log\sigma + b\log\sigma$ bits of space, instead of the (at least) $n\log\sigma + o(n)\log\sigma$ bits required by variants to date. If the final index does not have to reside in memory then at most $2b\log\sigma$ bits are needed for construction of the index from the BWT. Construction time remains linear and is fast in practice as the BWT is scanned only once, from left to right. For example, construction a fixed block index for the XML file takes just 12 seconds, while to build an index with optimal compression boosting requires 273 seconds. Ease of construction is also important when the aim is full inversion of the BWT in a general purpose file compressor [9].

References

1. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California (1994)
2. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In Amir, A., Moffat, A., Turpin, A., eds.: *Proc. 15th International Symposium on String Processing and Information Retrieval*. Volume 5280 of *LNCS*., Springer-Verlag (2008) 176–187
3. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. *Journal of the ACM* **52** (2005) 688–713
4. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics* **13** (2009) 1.12–1.31
5. Ferragina, P., Manzini, G.: Indexing compressed text. *Journal of the ACM* **52** (2005) 552–581
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms* **3** (2007) Article 20
7. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: *SODA '03: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2003) 841–850
8. Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: experiments with compressing suffix arrays and applications. In: *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2004) 636–645
9. Kärkkäinen, J., Puglisi, S.J.: Medium-space algorithms for inverse bwt. In de Berg, M., Meyer, U., eds.: *Proceedings of the 18th European Symposium on Algorithms (ESA '10)*. Volume 6346 of *LNCS*., Springer-Verlag (2010) 451–462
10. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing* **12** (2005) 40–66
11. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In Ziviani, N., Baeza-Yates, R.A., eds.: *Proc. 14th International Symposium on String Processing and Information Retrieval*. Volume 4726 of *LNCS*., Springer-Verlag (2007) 229–241
12. Manzini, G.: An analysis of the Burrows-Wheeler transform. *Journal of the ACM* **48** (2001) 407–430
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. *ACM Computing Surveys* **39** (2007)
14. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, SIAM (2007)
15. Raman, R., Raman, V., Rao-Satti, S.: Succinct indexable dictionaries with applications to encoding -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms* **3** (2007)
16. Vigna, S.: Broadword implementation of rank/select queries. In McGeoch, C.C., ed.: *Proc. 7th International Workshop on Experimental Algorithms (WEA 2008)*. Volume 5038 of *LNCS*., Springer-Verlag (2008) 154–168